Django PyNotify

Release 0.5.5

Ondřej Kulatý

Jan 03, 2023

CONTENTS:

CHAPTER

ONE

DJANGO PYNOTIFY

General purpose notification library for Django.

- Free software: MIT license
- Documentation: https://django-pynotify.readthedocs.io.
- Supported Python versions: 3.7, 3.8, 3.9, 3.10, 3.11
- Supported Django versions: 3.1, 3.2

1.1 Features

- Easy integration into project
- Notification templating and translation
- Asynchronous operation

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install PyNotify, run this command in your terminal:

```
$ pip install django-pynotify
```

This is the preferred method of installation, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/druids/django-pynotify
```

Or download the tarball:

```
$ curl -OL https://github.com/druids/django-pynotify/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ make install
```

2.3 Enable the library

Once installed, add the library to `INSTALLED_APPS` in your Django project settings:

```
INSTALLED_APPS = [
    ...
    'pynotify.apps.PyNotifyConfig'
]
```

CHAPTER

THREE

USAGE

3.1 Overview

The library uses following pipeline of components that handle creatation of notifications.

3.2 Signals

Notification creation process starts by sending a Django signal. This mechanism allows hooking notifications to various events within the system (even inside 3rd party libraries) and helps to decouple the code.

Following types of signals can be used:

- internal Django signals, like `post_save`
- singals from 3rd party libraries, like `password_set` from **django-allauth**
- your own signals

Imagine we would like to create a notification for an author, when someone reads his article. We start by defining a custom signal:

```
from django.dispatch import Signal

article_viewed = Signal(providing_args=['user', 'article'])
```

When this signal is sent, it means that `user` has viewed `article`.

3.3 Receivers

Receivers receive signals kwargs and pass them to handlers. There are currently two receivers implemented in the library:

- *SynchronousReceiver*
- *AsynchronousReceiver*

By default, the library is using the synchronous receiver, because it is easier for use. The synchronous receiver calls directly the handler. Asynchronous receiver is more efficient, but requires additional setup, see *Asynchronous operation*.

You can set which receiver will be used in the pipeline by changing `PYNOTIFY_RECEIVER` setting (see *Configuration*). If you need to implement your own receiver, you should inherit from *BaseReceiver*.

Most of the time, you don't need to implement your own receivers.

3.4 Handlers

Handlers are probably the most important part of the pipeline. They handle creating of *Notification* model instance(s) from signal kwargs received from a receiver.

Handler is typically the only thing you need to define in order to create a new type of notification, provided the signal you want to react upon already exists.

Let's create a simple handler for our "article viewed" notification:

```
from pynotify.handlers import BaseHandler
from articles.signals import article_viewed

class ArticleViewedHandler(BaseHandler):

    def get_recipients(self):
        return [self.signal_kwargs['article'].author]

    def get_template_data(self):
        return {
            'title': 'Your article has been viewed!',
        }

    class Meta:
        signal = article_viewed
```

As you can see, you need to implement at least following methods in the handler:

- *get_recipients()*
- *get_template_data()*

All handlers are typically kept in file `handlers.py` in a dedicated application within the project. When handler is defined (or more specifically, imported), it is paired with the signal defined in handler's `Meta`.

Let's say you created `notifications` app with `notifications/handlers.py`. In order for handlers to be automatically loaded, you must either set the `PYNOTIFY_AUTOLOAD_MODULES` in project settings:

```
PYNOTIFY_AUTOLOAD_MODULES = ('notifications.handlers',)
```

or load handlers module manually when Django is ready, i.e. put following code to `notifications/apps.py`:

```
from django.apps import AppConfig

class NotificationsConfig(AppConfig):

    name = 'notifications'
    verbose_name = 'Notifications'

    def ready(self):
        from . import handlers
```

Now, when you send the `article_viewed` signal, a new notification will be created for article author.

Note: It is possible to set `signal = None` in handler's `Meta`. In that case, the handler won't be paired with any signal and it's up to you to call it directly. There are two use cases for this feature:

1. You want to use some custom signal mechanism, bypassing Django signals completely
2. You want to process created notifications outside of the handler (they are returned by handler's `handle` method)

3.5 Templates

Templates are blueprints for notifications, they are referenced in the notification and are used to dynamically render notification fields. Handler's method *get_template_data()* returns values for *NotificationTemplate* attributes.

When notification is being created, handler first checks if template with attributes returned by *get_template_data()* exists. If not, the template is first created and then assigned to the created notification.

The most powerful feature of templates is probably the ability to dynamically render related objects. This can be best illustrated with an example. We will improve the "article viewed" notification from the previous section:

```
from pynotify.handlers import BaseHandler
from articles.signals import article_viewed

class ArticleViewedHandler(BaseHandler):

    def get_recipients(self):
        return [self.signal_kwargs['article'].author]

    def get_template_data(self):
        return {
            'title': '<b>{{user}}</b> viewed your article {{article}}',
            'trigger_action': '{{article.get_absolute_url}}'
        }

    def get_related_objects(self):
        return {
            'user': self.signal_kwargs['user'],
            'article': self.signal_kwargs['article']
        }

    class Meta:
        signal = article_viewed
```

As you can see, we have changed the template strings to true Django templates, because the template fields, when accessed through *Notification*, are rendered using Django template engine with context filled with named related objects. This is very convenient since notifications will always stay up to date, even if related object changes.

Note: For security reasons, you can only access related object's string representation and a set of attributes defined in `PYNOTIFY_RELATED_OBJECTS_ALLOWED_ATTRIBUTES`. See *Configuration* for more information.

Caution: Avoid adding unnecessary attributes to `PYNOTIFY_RELATED_OBJECTS_ALLOWED_ATTRIBUTES`, since it increases coupling between notification template(s) and the code. This is undesirable and makes managing and maintenance of notifications harder.

Always consider first to store attribute's value in extra data (as described lower), or save nested objects as standalone related objects (if you really need dynamic behavior).

In case you want to "freeze" the values used in template strings (i.e. not reflect changes made in related objects), define `get_extra_data()`, which should return a dict of JSON serizalizable values. These extra data are also put into template context, together with named related objects.

If you need some extra fields, that are relevant to your use case, you can pass `extra_fields`, which is expected to a be a flat dictionary of strings. These extra fields are also dynamically rendered, just like standard notification fields.

Instead of using *get_template_data()*, you can define handler's attribute `template_slug`. This is a better option in case you prefer to often change template strings via administration interface. Note, that the admin template (*AdminNotificationTemplate*) referenced by slug must already exist - it won't be automatically created. You can create it in administration interface or using data migration.

Given the admin template with slug *article-viewed*, our handler can be modified as follows:

```
from pynotify.handlers import BaseHandler
from articles.signals import article_viewed

class ArticleViewedHandler(BaseHandler):

    template_slug = 'article-viewed'

    def get_recipients(self):
        return [self.signal_kwargs['article'].author]

    def get_related_objects(self):
        return {
            'user': self.signal_kwargs['user'],
            'article': self.signal_kwargs['article']
        }

    class Meta:
        signal = article_viewed
```

3.6 Dispatchers

Dispatchers are used by handlers to propagate notifications through various communication channels, e.g. SMS, e-mails, push. The library currently does not include any specific dispatchers, just the base class *BaseDispatcher*.

Let's implement e-mail notifications for our "article viewed" notification. We'll start by creating an e-mail dispatcher:

```
from pynotify.dispatchers import BaseDispatcher
from django.core.mail import send_mail

class EmailDispatcher(BaseDispatcher):

    def dispatch(notification):
```

(continues on next page)

(continued from previous page)

```
        send_mail(
            subject=notification.title,
            message=notification.text,
            from_email='noreply@example.com',
            recipient_list=(notification.recipient.email,),
        )
```

And now we will add our dispatcher to the handler:

```
from pynotify.handlers import BaseHandler
from articles.signals import article_viewed

from .dispatcher import EmailDispatcher

class ArticleViewedHandler(BaseHandler):

    dispatchers = (EmailDispatcher,)

    ...
    ...
```

CHAPTER

FOUR

EXTRA TIPS

4.1 Simplified usage

If all you need is just to create notifications at some point in your code, you can start doing so right away after installation of the library. No further configuration needed!

To create a notification, simply call the *notify()* function:

```
from django.contrib.auth.models import User
from pynotify.notify import notify

notify(recipients=User.object.all(), title='Hello World!')
```

Even with this simple approach, you can still use templating and/or translations.

4.2 Translations

Notification templates can be translated using the standard Django translation mechanism. The only thing needed is to enable template translation by setting `PYNOTIFY_TEMPLATE_TRANSLATE` to `True` and include the translated messages in `*.po` files.

You can use `gettext_noop()` when defining template strings, so the string will be automatically included in the transaltion file(s):

```
def get_template_data(self):
    return {
        'title': gettext_noop('{{user}} viewed your article {{article}}'),
    }
```

However keep in mind, that if you change the template string inside `gettext_noop()`, you either have to change the corresponding notification template saved in the database (e.g. using data migration) or keep the old string in the translation file.

In case you are using **template slugs**, just put `gettext_noop()` anywhere in the code and keep it in sync with contents of the notification template saved in the database:

```
class ArticleViewedHandler(BaseHandler):

    template_slug = 'article-viewed'
```

(continues on next page)

(continued from previous page)

```
    # title translation
    gettext_noop('{{user}} viewed your article {{article}}')
```

4.3 Asynchronous operation

Creating notifications can be time demanding, especially when creating a lot of notifications at once or dispatching via 3rd party services (e.g. SMS, e-mails, push). Using the default synchronous operation in these cases considerably extends time needed to process a request. Therefore, **it is recommended to always switch to asynchronous mode, if you can.**

The library contains *AsynchronousReceiver*, which allows asynchronous operation. Intead of calling a handler directly, it works by passing serialized signal kwargs to a Celery task upon database transaction commit and the task then calls the handler. Since serialization comes into play here, signal kwargs are restricted to be either directly JSON serializable values or model instances (which are serialized using built-in *ModelSerializer*).

To go asynchronous, change setting `PYNOTIFY_RECEIVER` to `pynotify.receivers.AsynchronousReceiver` and start Celery in your project in autodiscover mode. See http://docs.celeryproject.org/en/latest/django/first-steps-with-django.html.

The Celery task is defined in the library, you don't have to create one. But in case you want to use a custom Celery task, set its import path to `PYNOTIFY_CELERY_TASK` setting. Your custom task should grab all the arguments it receives and pass them to *process_task()*, like this:

```
from pynotify.helpers import process_task

@shared_task
def my_task(*args, **kwargs):
    process_task(*args, **kwargs)
```

CHAPTER

FIVE

CONFIGURATION

You can configure the library in Django settings. Following options are available:

- `PYNOTIFY_AUTOLOAD_MODULES` (default: `None`)

  Iterable of Python modules that contain notification handlers. These modules will be imported at startup, i.e. causing notification handlers to be automatically registered. For example, if you have Django app `notifications` with handlers stored in `handlers.py`, the module for autoload will be `notifications.handlers`.

- `PYNOTIFY_CELERY_TASK` (default: `pynotify.tasks.notification_task`)

  Import path to a Celery task used in asynchronous mode. See *Asynchronous operation*.

- `PYNOTIFY_ENABLED` (default: `True`)

  Boolean indicating if library functionality, i.e. creating of notifications, is enabled. More specifically, if set to `False`, receiver will not be called upon signal reception.

- `PYNOTIFY_RECEIVER` (default: `pynotify.receivers.SynchronousReceiver`)

  Import path to a receiver class.

- `PYNOTIFY_RELATED_OBJECTS_ALLOWED_ATTRIBUTES` (default: `{'get_absolute_url', }`)

  A set of related object's attributes that can be used in notification template(s).

- `PYNOTIFY_STRIP_HTML` (default: `False`)

  If set to `True`, HTML tags and entities will be stripped off during notification rendering.

- `PYNOTIFY_TEMPLATE_CHECK` (default: `False`)

  Boolean indicating if template string should be checked before rendering. If any named related object or extra data used in the template string is missing, *`MissingContextVariableError`* will be raised.

- `PYNOTIFY_TEMPLATE_PREFIX` (default: `''`)

  String that is prepended to any template just before rendering. Can be used to load custom tags/filters.

- `PYNOTIFY_TEMPLATE_TRANSLATE` (default: `False`)

  Boolean indicating if template string should be translated via `gettext()` before rendering.

CHAPTER

SIX

EXAMPLE PROJECT

You can quickly try the library in the included example project. Install the library for development, as described in *Contributing* and run following commands:

```
$ cd example
$ ./manage.py runserver
```

The example project will be available at https://localhost:8000/.

6.1 Asynchronous operation

If you want to try *Asynchronous operation* in the example project, make sure you have Redis installed and uncomment following settings in `example/config/settings.py`:

```
CELERY_BROKER_URL = 'redis://127.0.0.1'
PYNOTIFY_RECEIVER = 'pynotify.receivers.AsynchronousReceiver'
```

Then open a new terminal window and start Celery with:

```
$ cd example
$ celery -A config worker
```

CHAPTER

SEVEN

REFERENCE

7.1 pynotify.config

class pynotify.config.**Settings**

Bases: object

Holds default configuration values, the values can be overriden in settings with PYNOTIFY_ prefix.

7.2 pynotify.dispatchers

class pynotify.dispatchers.**BaseDispatcher**

Bases: object

Base class for sending notification over a communication channel (e.g. e-mail, sms, push).

dispatch(*notification*)

This method should implement actual sending of notification.

7.3 pynotify.exceptions

exception pynotify.exceptions.**MissingContextVariableError**(*field_name*, *variable*)

Bases: Exception

Raised when template field cannot be rendered because variable used in it is not present in the context.

7.4 pynotify.handlers

class pynotify.handlers.**BaseHandler**

Bases: object

Base class for handling creation of notification(s). Its purpose is to process signal kwargs sent over a defined signal. There should be typically one handler (inherited from this class) for each signal. The handler must define inner class Meta with following supported attributes:

- signal: Signal to which handler will be registered
- allowed_senders: Handler will be called only if signal was sent by allowed sender

- `abstract`: If set to True, the handler will not be registered

dispatcher_classes

An iterable of dispatcher classes that will be used to dispatch each notification.

template_slug

Slug of an existing admin template to be used. If not defined, you must define `get_template_data()` method.

get_dispatcher_classes()

Returns iterable of dispatcher classes used to dispatch notification(s).

get_extra_data()

Returns a dictionary with extra data, the values must be JSON serializable.

get_recipients()

Returns an iterable of recipients for which notification will be created.

get_related_objects()

Returns a list or dictionary of related objects in format {"name": object}. Named related objects (i.e. those passed using a dictionary) can be referred in notification template.

get_template_data()

Returns kwargs used to create a template. Not called if template slug is used.

get_template_slug()

Returns slug of an admin template to be used.

handle(*signal_kwargs*)

Handles creation of notifications from `signal_kwargs`.

class pynotify.handlers.**HandlerMeta**(*name*, *bases*, *attrs*)

Bases: `type`

Registers handler for handling of signal defined in handler's `Meta`.

7.5 pynotify.helpers

class pynotify.helpers.**DeletedRelatedObject**

Bases: `object`

Placeholder class that substitutes deleted related object and returns:

- "[DELETED]" as its string representation
- itself for any attribute accessed

class pynotify.helpers.**SecureRelatedObject**(*related_object*)

Bases: `object`

Security proxy class allowing to access only string representation of the related object and a set of attributes defined in *RELATED_OBJECTS_ALLOWED_ATTRS* settings.

class pynotify.helpers.**SignalMap**

Bases: `object`

Maps signals to arbitrary values.

pynotify.helpers.**autoload**()

Attempts to load (import) notification handlers from modules defined in PYNOTIFY_AUTOLOAD_MODULES

pynotify.helpers.**get_from_context**(*variable*, *context*)

Tries to find *variable* value in given *context*.

Parameters

- **variable** – Variable to look for. Template format is supported (e.g. "abc.def.ghi").
- **context** – Template context.

Returns

Variable value or None if not found.

pynotify.helpers.**get_import_path**(*_class*)

Returns import path for a given class.

pynotify.helpers.**process_task**(*handler_class*, *serializer_class*, *signal_kwargs*)

Deserializes signal kwargs using the given serializer and calls given handler. This function is intended to be called from a Celery task.

pynotify.helpers.**receive**(*sender*, ***kwargs*)

Initiates processing of the signal by notification handlers through a receiver.

pynotify.helpers.**register**(*signal*, *handler_class*, *allowed_senders=None*)

Starts listening to `signal` and registers `handler_class` to it.

pynotify.helpers.**strip_html**(*value*)

Strips HTML (tags and entities) from string *value*.

7.6 pynotify.models

class pynotify.models.**AdminNotificationTemplate**(**args*, ***kwargs*)

Bases: *BaseTemplate*

Represents a "template of a template". This model is intended to be managed from administration, hence its name. It is identified by *slug*, which can be used for notification creation. However, this template is never used to directly render a notification, but instead is used to create *NotificationTemplate* with same values.

slug

Template slug, with which this template can be referred to.

is_active

Flag that switches on/off creating notifications from this template.

is_locked

Flag that switches on/off this template editing (for admin purposes, requires admin-side support).

send_push

Flag that switches on/off sending push notifications from this template. Currently, it has no effect on its own, but you can use it in your custom push notification solution.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

class pynotify.models.**BaseModel**(**args*, ***kwargs*)

Bases: SmartModel

Base class for models that outpus its verbose name and PK.

class pynotify.models.**BaseTemplate**(**args*, ***kwargs*)

Bases: *BaseModel*

Base abstract model for notification template.

title

Title of the notification.

text

Text of the notification.

trigger_action

Arbitrary action performed when user triggers (i.e. clicks/taps) the notification.

extra_fields

Can be used to store additional fields needed in particular use case.

class pynotify.models.**Notification**(**args*, ***kwargs*)

Bases: *BaseModel*

Represents the notification.

Attributes specified in TEMPLATE_FIELDS are also available here, as generated properties, that are evaluated at runtime and will return rendered field from the associated template. By default, the context used for rendering is filled with named related objects and extra data, so they can be referenced in the template by their name/key.

recipient

Recipient of the notification.

template

Template used to render generated notification fields.

is_read

Boolean flag indicating that recipitent has seen the notification.

is_triggered

Boolean flag indicating that recipient has triggered the notification (e.g. clicked/tapped)

extra_data

JSON serialized dictionary with extra data.

exception DoesNotExist

Bases: ObjectDoesNotExist

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

clean()

Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

property context

Returns context dictionary used for rendering the template.

related_objects_dict

Returns named related objects as a dictionary where key is name of the related object and value is the object itself. Related objects without name are skipped.

class pynotify.models.**NotificationMeta**(*name*, *bases*, *attrs*)

Bases: SmartModelBase, type

Creates property for each template field. The property returns rendered template.

class pynotify.models.**NotificationQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Bases: SmartQuerySet

create(*recipient*, *template*, *related_objects=None*, ***kwargs*)

Create a new object with the given kwargs, saving it to the database and returning the created object.

class pynotify.models.**NotificationRelatedObject**(**args*, ***kwargs*)

Bases: *BaseModel*

Represents object related to a notification. This object can be then referenced in notification template fields by its *name* (if not None).

name

String identificator of the object (for referencing in templates).

notification

Related notification.

content_object

The related object itself.

exception DoesNotExist

Bases: ObjectDoesNotExist

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

class pynotify.models.**NotificationTemplate**(**args*, ***kwargs*)

Bases: *BaseTemplate*

Represents template that is used for rendering notification fields. Each field specified in TEMPLATE_FIELDS is a template string, that can be rendered using the render method.

admin_template

Reference to admin template that was used to create this notification template.

exception DoesNotExist

Bases: ObjectDoesNotExist

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

render(*field*, *context*)

Renders field using context.

7.7 pynotify.notify

class pynotify.notify.**NotifyHandler**

Bases: *BaseHandler*

Notification handler for the `notify` method.

get_dispatcher_classes()

Returns iterable of dispatcher classes used to dispatch notification(s).

get_extra_data()

Returns a dictionary with extra data, the values must be JSON serializable.

get_recipients()

Returns an iterable of recipients for which notification will be created.

get_related_objects()

Returns a list or dictionary of related objects in format {"name": object}. Named related objects (i.e. those passed using a dictionary) can be referred in notification template.

get_template_data()

Returns kwargs used to create a template. Not called if template slug is used.

get_template_slug()

Returns slug of an admin template to be used.

pynotify.notify.**notify**(*recipients*, *related_objects=None*, *extra_data=None*, *template_slug=None*, *dispatcher_classes=None*, ***template_fields*)

Helper method to simplify notification creation without the need to define signal and handler.

7.8 pynotify.receivers

class pynotify.receivers.**AsynchronousReceiver**(*handler_class*)

Bases: *BaseReceiver*

Signal receiver that calls notification handler asynchronously via Celery.

receive(*signal_kwargs*)

This method should implement passing `signal_kwargs` to the handler.

class pynotify.receivers.**BaseReceiver**(*handler_class*)

Bases: `object`

Base class for receiving signals. Its purpose is to pass signal kwargs to the notification handler.

receive(*signal_kwargs*)

This method should implement passing `signal_kwargs` to the handler.

class pynotify.receivers.**SynchronousReceiver**(*handler_class*)

Bases: *BaseReceiver*

Signal receiver that calls notification handler synchronously.

receive(*signal_kwargs*)

This method should implement passing `signal_kwargs` to the handler.

7.9 pynotify.serializers

class pynotify.serializers.**BaseSerializer**

Bases: object

Base class for serializing/deserializing signal kwargs. Its puprose is to transform signal kwargs to be directly JSON serializable (for compatible types, see https://docs.python.org/3/library/json.html#py-to-json-table).

deserialize(*signal_kwargs*)

This method should return deserialized signal_kwargs.

serialize(*signal_kwargs*)

This method should return serialized signal_kwargs.

class pynotify.serializers.**ModelSerializer**

Bases: object

Serializes any model instance into its PK and ContentType PK and deserializes by fetching the model instance from database. Works recursively on nested dicts and iterables. Values that are not model instances are left intact.

7.10 pynotify.tasks

pynotify.tasks.**notification_task**(*self*, **args*, ***kwargs*)

Celery task used in asynchronous mode. Just passes any arguments to process_task function.

CHAPTER

EIGHT

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

8.1 Types of Contributions

8.1.1 Report Bugs

Report bugs at https://github.com/druids/django-pynotify/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

8.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

8.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

8.1.4 Write Documentation

Django PyNotify could always use more documentation, whether as part of the official Django PyNotify docs, in docstrings, or even on the web in blog posts, articles, and such.

8.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/druids/django-pynotify/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

8.2 Get Started!

Ready to contribute? Here's how to set up **Django PyNotify** for local development.

1. Fork the repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-pynotify.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-pynotify
$ cd django-pynotify/
$ make develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

   To create migrations, run:

```
$ make makemigrations
```

   To make translations, run:

```
$ make po
```

   To compile translations, run:

```
$ make mo
```

5. When you're done making changes, check that your changes pass flake8 and the tests:

```
$ make lint
$ make test
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst. Also consider inplementing the new functionality in the example project.
3. The pull request should work for all supported Python versions, and for PyPy. Check https://travis-ci.org/druids/django-pynotify/pull_requests and make sure that the tests pass for all supported Python versions.

8.4 Tips

To run a subset of tests:

```
$ cd example
$ ./manage.py test tests.test_config
```

8.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push && git push --tags
$ make release
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

NINE

CREDITS

List of people who helped with development of this library. Feel free to add your name to the list, if you've made a contribution.

9.1 Development Lead

- Ondřej Kulatý <kulaty.o@gmail.com>

9.2 Contributors

- Luboš Mátl
- Petr Olah

CHAPTER

TEN

HISTORY

10.1 0.5.5 (2023-01-03)

- Add support for Python 3.10 and 3.11

10.2 0.5.4 (2022-03-10)

- Migrate CI from Travis to Github Actions
- Fix lint errors

10.3 0.5.3 (2022-03-01)

- Prevent duplicates of `NotificationTemplate` objects

10.4 0.5.2 (2022-02-21)

- Allow handlers that are not paired with any signal
- Return notifications created with `notify` helper function

10.5 0.5.1 (2022-01-20)

- Use `JSONField` instead of `TextField` for JSON based fields
- Add `extra_fields` to `BaseTemplate`
- Drop Django 2.x support

10.6 0.4.6 (2021-08-31)

- Add `is_locked` field to `AdminNotificationTemplate`

10.7 0.4.5 (2021-01-21)

- Update dependencies

10.8 0.4.4 (2021-01-15)

- Add support for Python 3.9
- Add support for Django 3
- Fix BS4 warning

10.9 0.4.3 (2020-12-16)

- Fix translation file

10.10 0.4.2 (2020-12-11)

- Add `send_push` flag to `AdminNotificationTemplate` model
- Ignore duplicit dispatcher classes in `BaseHandler`

10.11 0.4.1 (2020-10-12)

- Add `PYNOTIFY_STRIP_HTML` config option

10.12 0.4.0 (2020-08-12)

- Removed support of Django 1.11, 2.0 and 2.1
- Fixed library requirements

10.13 0.3.2 (2020-07-27)

- Add `is_active` flag to `AdminNotificationTemplate` model

10.14 0.3.1 (2020-06-12)

- Improve template variable checking
- Add new filter `filter_with_related_object`

10.15 0.3.0 (2020-04-19)

- Fix documentation
- Change `PYNOTIFY_AUTOLOAD_APPS` to `PYNOTIFY_AUTOLOAD_MODULES`, i.e. allow notification handlers to reside in arbitrary module

10.16 0.2.2 (2020-02-11)

- Use Django JSON encoder for encoding extra data

10.17 0.2.1 (2020-02-11)

- Fix failed PyPi upload

10.18 0.2.0 (2020-02-11)

- Add admin templates
- Limit usage of related objects in templates and add `PYNOTIFY_RELATED_OBJECTS_ALLOWED_ATTRIBUTES` setting
- Show placeholder text for deleted related objects

10.19 0.1.7 (2020-01-20)

- Add support for Python 3.8 and Django 2.2
- Fix generating of translations
- Allow unnamed related objects to be passed in a list

10.20 0.1.6 (2019-04-16)

- Add `PYNOTIFY_TEMPLATE_PREFIX` config option
- Add methods `get_template_slug()` and `get_dispatcher_classes()` to `BaseHandler`
- Add coveralls.io integration

10.21 0.1.5 (2019-04-12)

- Add extra data to `Notification` model

10.22 0.1.4 (2019-04-08)

- Add `_can_handle()` method to `BaseHandler`
- Add `PYNOTIFY_ENABLED` setting

10.23 0.1.3 (2019-04-01)

- Add `kwargs` to Notification manager's `create()` method
- Add `realted_objects_dict` property to `Notification` model

10.24 0.1.2 (2019-03-20)

- Remove automatic deploy to PyPi from Travis

10.25 0.1.1 (2019-03-20)

- First release of the library

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

p

INDEX

M

N

P

R

S

T